Load Balancing by Tailoring Programs for Migration

Andrew Stewart

Honours project, 1994

Abstract

A number of approaches have been taken to the problem of load balancing, most of these centred around the concepts of placement and migration of *arbitrary* (hopefully compute-intensive) processes. These techniques can be difficult to retro-fit into existing systems. In this report we introduce the concept of migration of *selected* "work generating" processes. By specifically modifying an application migration can be achieved more easily than by other methods. We describe a prototype system developed at the University of Canterbury, consisting of a version of `tcsh` that attempts to balance future sources of work by migrating to lightly loaded processing nodes. Finally some of the potential benefits and drawbacks of this approach are considered.

Contents

Chapter 1

Introduction

Over the past few years the power of computing technology has increased dramatically. At the same time, the cost of that technology has been falling, so that relatively powerful computers are now commonplace. The potential benefits of interconnecting these computers to allow sharing of resources and computing power has become increasingly evident.

Some of these benefits have been partially realised by the now common use of Network Operating Systems, systems that consist of a number of machines (processing nodes) linked together by a network. The computers in a network operating system are still largely autonomous, and this can make sharing of processing power difficult.

A newer development, the Distributed Operating System, promises to allow the integration of the computers on a network so that they together form a larger *virtual* machine, achieved through the sharing of processing work throughout the processing nodes on the network. At the current state of developments, such distributed operating systems are still relatively rare, and most are developmental systems.

The transition to full distributed operating systems is not an easy one to make. Retaining compatibility with existing operating systems is not ensured, and it is difficult to alter these existing systems to take full advantage of the potential of a distributed operating system. Instead, a number of attempts have been made to work with existing network operating systems without altering them, to derive some of the benefits of a fully distributed operating system. In particular a benefit we are interested in is load balancing, sharing the processing workload amongst processing nodes so that collectively they will respond more quickly to user demands.

Most of the attempts to provide load balancing centre around some of the techniques used in many distributed operating systems. These are:

- Process migration. An already running program (process) is taken from one processing node and transferred to another processing node. Migration of arbitrary processes is difficult to 'retro-fit' into existing operating systems because it is difficult to ensure that a migrated process will continue to work in its new location.

- Process placement. When a new program is started execution of that

program may take place on a remote machine. This method has its own drawbacks — these are discussed later.

1.1 Tailoring Processes for Migration

To effectively balance the workload across all of the processing nodes in a network, it is unnecessary to make all of the workload in the system mobile and capable of migration. Rather, only the *excess* work at overloaded nodes needs to be mobile, so that it may be moved and shared out across the other processing nodes.

Because of this, we can use a new method of load balancing that is simpler to implement than migration of arbitrary processes, but will not have some of the costs of process placement.

This method is the alteration or tailoring of processes for migration. Selected 'work-generating' programs, those that contribute in a significant manner to the overall load on a system, are modified to allow them to be easily migrated across a network operating system. Modifying programs for migration allows us to have a form of load balancing that gets around many of the difficulties associated with process migration, and at the same time avoid most of the drawbacks of process placement.

1.2 The Migrating Shell

The aim of the project described in this report has been to investigate some of the issues involved in the application of the migration tailoring concept, and to assess the overall merit of the method.

This has been done by implementing a prototype load balancing system that uses this method, a migrating version of the `tcsh` command interpreter. It has been possible to gain insights from the construction of and experimentation with the new system.

1.3 Structure of This Report

The next chapter, chapter 2 is an introduction to many of the issues that are relevant to load balancing and the migration tailoring method in particular. A number of other load distributing systems that have been implemented in the past are also surveyed.

Chapter 3 is an introduction to the tailored migration approach. The general principles behind how the migrating shell works are discussed, as well the reasoning behind its consideration as a valid load balancing approach.

In chapter 4 we look at the implementation of the two parts that comprise the system. We also consider some of the problems encountered and potential solutions to these problems. We then consider the viability and future of the tailored migration concept as a whole in chapter 5. Finally, some conclusions are presented in chapter 6.

Chapter 2

Load Balancing Approaches

The area of load distribution has been the focus of much research over the past ten years. A number of different approaches have been taken to tackling the problem, and a number of important areas of interest have been identified. Some of the work and issues more closely related to the area of load balancing are discussed here, and further information is also available (Tanenbaum, 1992; Goscinski, 1991; Hac, 1989).

2.1 Load Balancing Mechanisms

Something all load balancing schemes have in common is a requirement for the use of some form of migration mechanism, a system that facilitates the movement of work from one processing node to another. Load balancing schemes will use one or more of the following load distribution mechanisms.

2.1.1 User Session Placement

The placement of user login sessions is a relatively new concept introduced in the CLB system (Smith, 1992; Ashton & Smith, 1993). Rather than placing individual processes, the system balances load across the processing nodes on the network by placing entire login sessions. As each user logs in, they are placed on the most lightly loaded machine on the network. Thereafter any work the user generates appears on the machine the user has been placed on.

This method of load balancing is one without many of the problems of process placement and process migration below. In particular, if running in an X-windows environment as it is designed to do, there are almost no additional costs associated with remote execution and transparency issues in the CLB system. These factors make it an attractive option even in poor conditions when the return from it is small.

2.1.2 Process Placement

In a process placement system, processes are allocated to machines as they are created. There is a cost in arranging for these processes to run on the other

machine: the process must be transferred and a new context set up on the destination machine.

Because of this cost it is not worthwhile starting up short processes on remote machines — it might take more time to transfer the process than it would to run the process locally! Process placement, by definition, must act at the time when a new process is about to start. A problem with acting at this point is that no past information on this particular process is available, and therefore cannot be used to tell whether a process will be long lived (so worth migrating). Systems get around this problem by keeping some form of information about the command being executed. This might take the form of a configuration file as in (Ferrari & Zhou, 1988), which contains a list of commands that the system knows may be executed remotely.

The time cost of process placement also has an impact on the transparency of the system. Ideally a user should not notice that their process is being run remotely. Setting up the remote execution takes several seconds, so setup time is likely to be a problem. The system developed by (Ferrari & Zhou, 1988) combats this to some extent by doing part of the remote execution setup beforehand so that future remote executions take less time.

2.1.3 Process Migration

Process migration involves taking a process that is running on an overloaded machine and moving that process to another machine, where it can resume execution.

This is a complex and resource intensive operation, more so than process placement, but it does help to ensure that effort is not expended on short-lived processes: some of the past characteristics of a process (for instance processor time used) can be taken into account when considering candidate processes for migration (Barak & Shiloh, 1985).

Process migration can also improve the ability of the system to cope with changes in the loads of machines, due to the fact that it can move work at any time (although there is some debate as to whether this actually yields a significant improvement in performance) (Ferrari & Zhou, 1988; Eager *et al.*, 1988).

2.1.4 Granularity of Migration

The above techniques represent a cross section of possible trade-offs in terms of transparency, computational effort and importantly from our point of view, granularity.

The granularity of a load moving system is the unit by which load movement is performed. A system with a finer granularity has more flexibility over the control it can apply in the load balancing procedure. There are costs associated with increased granularity too, and these are discussed in more detail here.

The load moving mechanisms described above have been ordered in terms of granularity, from coarse granularity in the case of CLBs user placement strategy, to fine granularity in the case of a process migration system.

CLBs migration mechanism can be described as coarse because the smallest unit of workload CLB can deal with is an entire user login session. The amount of work a user may generate in a session of work placed in this way is potentially very large. Also the method employed by CLB only allows it to adjust the load relatively infrequently. As a result of its coarse granularity, the effectiveness of CLB can be diminished if a steady stream of user logins does not occur. CLB may also take longer to respond to large changes in processing node load levels.

A process placement method offers a finer granularity than a user session placement method. Relatively speaking, process creations occur more often than user logins, so more opportunities for load moving are available. This increased granularity is not free though, as the use of process placement methods bring their own set of problems, as mentioned above.

The finest granularity and the greatest potential for control is offered by the process migration methods. Correspondingly, inherent in these methods is also the highest cost.

2.2 Load Balancing Policy

It is important not to confuse the load distribution mechanism, described above, with distribution policy. The mechanism is that part of the system that moves the load from one machine to another; the policy is the part of the system that decides how the mechanism will be used.

There are three points the policy must address:

- What work to move. The policy must decide which parts of the load must be moved. This is important because it is not desirable to move some parts of the load. For instance, it is inefficient to remotely execute or migrate a very short-lived process: a machine might do more work moving a process than it would if it simply completed execution of the process locally.

- Where to move the work. The load must be moved to a place that will improve the balance across a network of processing nodes. If we are moving several parts of the load around at once, we need to guard against problems like 'flocking' of processes. An example of this is where a number of processing nodes simultaneously decide to offload work onto one lightly loaded node. At the time the independent decisions are made the node is lightly loaded, but it can become heavily overloaded if too much work is moved onto it.

- When to move the work. Periodically the system must consider the possibility of moving work. How often this is done can affect the performance of the system: if too often the load balancing system will itself begin to consume an excess of computing resources, and if not often enough, the system will tend to fall out of balance more easily.

A number of other policy issues are present in the implementation of a load balancing system. While these are important issues and the focus of much

research they are not of central interest in this report so they will only be covered briefly here.

Other major policy issues include:

- A load balancing policy is either optimal or sub-optimal. An optimal policy is one that knows in advance the amount of work that is to be generated by any process run within it, and uses this information for allocation of work to processing nodes. Few applications exist for which this information is available, so instead most load balancing systems are sub-optimal. A sub-optimal system uses observations about the system state (for instance load metrics) to make policy decisions. It watches the processing node loads and attempts to take corrective action whenever the system falls out of balance, as shown by the metric.

- Load movement can be either initiated by overloaded nodes (an overloaded node sends requests to other stations searching for places to offload its work to), or it can be initiated by idle processing nodes (the idle node sends requests for busy nodes to give it more work to do). The different types of initiation can be more or less suitable in different situations (Eager *et al.*, 1986).

- Load information can be distributed or centralised. In a centralised system all loading information is stored at some central, easily accessible point. In a distributed load information system there may be no one node with full load information for the whole network of processing nodes. The centralised method is easier to implement, and allows full load information to be easily collected by any node that requires it; however the central repository can become a bottleneck if many requests are made of it and a reliability weak point should it fail. Distributed information suffers from neither of these problems but is harder to implement and obtain data from (Tanenbaum, 1992)

2.3 Some Load Distributing Systems

A number of load distributing systems have been developed. Generally any one of these will fall into one of one of two categories (although some hybrids exist).

2.3.1 Utilising Idle Workstations

The first group is those that utilise processing nodes when they would otherwise be idle. Typically this is in a workstation environment, where each workstation on the network is dedicated to a single 'owner'. If this owner is not currently using the workstation (say nobody is logged in) then the workstation becomes a candidate for remote execution. Other users on their own workstations designate jobs to be executed on the idle workstations. Typically these 'farmed out' jobs were intended to be non-interactive intensive jobs (say compilations or simulations), but some implementors found users liked to use their system

for interactive programs even more (for instance mail or news readers) (Nichols, 1987).

These systems normally consider the 'owner' of the workstation to have an overriding priority over any other users' processes running on that workstation. Any processes currently running on that machine are typically 'evicted' when the owner returned. In some systems (for instance Butler (Nichols, 1987)) this involves killing the guest processes off. This has disadvantages in that any work done by those processes may be lost. To counter this, systems like Condor (Bricker *et al.*, 1992) use 'checkpointing' methods to stop progress being lost. This entails saving the entire state of a process periodically, so that if the process is killed it can be restarted from a checkpoint rather than starting over.

Other systems, for instance V (Theimer *et al.*, 1985), will allow guest processes to migrate away from a host when its owner returns, rather than kill them. This has the advantage that no work is lost, and that no work is performed twice (which is important in some applications). However all of the problems associated with migration appear, for instance resolution of file access handles and other transient identifiers on the new machine, and packaging and transport of all data and associated state information for the job. Migration also imposes a load on the machine from which the job is migrating at an inconvenient time (the owner has just returned and wants to use it, after all).

2.3.2 The Processor Pool

Other systems consider processing nodes to be ownerless — the jobs of any user of the system may run on any node with equal priority. On these systems process placement in isolation becomes viable for general processing use. This is because it is no longer necessary to deal with the problem of evicting processes from processing nodes at short notice.

The concept of a processor pool defines the pool as being a collection of one or more processors that are intended primarily for the purpose of serving the computing needs of remote users. Strictly speaking, the processor pool is expected to service only remote users. However for our purposes this model also suits our intended usage of the hybrid model, where users may also use the machines in the processor pool directly while these machines are participating as a part of the processor pool. The hybrid model is the organisation in place here at the University of Canterbury: many of the primary processing nodes have 'owners', but most are also available for remote execution.

2.4 Environment Transparency

Transparency is a major issue in distributed computing systems. Most programs people write are designed to work largely in a single-threaded manner, in a consistent environment, such as that provided by a single machine. When an attempt is made to run one of these programs on a remote processing node the program must be provided with the environment it would get were it running on its normal host.

The environment contains a number of elements that ideally should be maintained:

- The file system environment. When a program running on a remote machine makes a reference to a file, it is referring to that file as the file appears to the machine for which the program was intended. It must be provided with that file or transparency will be compromised. Most UNIX machines are normally mounted on completely different file systems — a reference to a filename on one machine is completely different to a reference to that same filename on another machine. File and filename transparency can be achieved to some extent using systems such as NFS or Andrew. However, neither of these file systems will perform as expected in cases where programs on two different processing nodes open the same file simultaneously.

  Other problems occur with file handles in migrated or checkpointed programs. A program might run for a time on one processing node (say node A) before being migrated to another processing node (node B). While running on node A, before migration, the program opens a file for reading, at which time the program is given a handle which is used thereafter to identify that file opening. When the program migrates to node B, the handle the program has been using no longer has meaning on the programs new host, since that handle refers to a file opened on node A. As a result, many migration systems will reopen the file at node B (at which time a new and different file handle is generated). Then the system must intercept any references to the node A file handle made by the program and translate them to the node B file handle before passing them on for processing.

- The user interaction element of the environment can be difficult to maintain. For even the simplest program, it is desirable to have at least some form of character interaction between the remote processing node where a job is running and the local node where the user interface is focussed. The program running on the remote node is likely to make standard input and output calls, and the results of these should be propagated to the local node.

  Simple character stream output is reasonably simple to provide, and is perfectly suitable for debugging or error reports. This support can be provided by the system to the user by writing program output text directly to the user's terminal (as opposed to providing a proper terminal output). Systems providing this kind of interface lose a lot of transparency, but are nevertheless still useful for batch-type jobs, for instance compilations or lengthy simulations.

  Providing a full terminal interface to the remote program is more difficult. Unfortunately this is what is necessary for many programs to operate correctly. As a result, for good transparency it is desirable to provide terminal propagation. This topic is described in some depth in (Stevens, 1990).

Some aspects of user interaction do work well from a remote environment. In particular, programs working within the X-windows environment can typically run remotely without special intervention regarding terminal propagation. This is because the ability to access remote displays (eg X-terminals) is built into the windowing system.

- The underlying system type and architecture also have an effect on a program running remotely. This is the issue of homogeneity. A program may be unlikely to run on a remote node if it is configured differently to the node the program was intended to run on. This includes cases where the two nodes' hardware is different or where the operating systems being run by the two nodes differ.

  Some systems (eg Amoeba, described in (Tanenbaum, 1992)) have actually been designed with the explicit heterogeneity possibility in mind. These will arrange for jobs to be run only on or moved to those machines that can support a compatible environment.

Chapter 3

Migrating Applications

3.1 An Alternative Approach to Load Balancing

As previously discussed, most systems implemented have been based around the process placement and migration mechanisms. These systems suffer primarily from the complexity of their solutions, and also from the additional work required in the migration or placement of arbitrary processes. CLB on the other hand does not suffer these problems, but is not always fine-grained enough in its control to effectively manage the load balancing process.

A compromise might exist, however, in the possibility of placing and migrating *specific processes* rather than arbitrary processes or entire login sessions.

3.1.1 The Migration Mechanism

An important point to consider with granularity as described in chapter 2 is precisely how much granularity is required. One study (Eager *et al.*, 1988) suggests that the fine-grained control permitted by process migration systems are not necessary in the implementation of an effective load balancing system. Experience with CLB has demonstrated that while the CLB system does an adequate job of balancing the workload, it may benefit from some form of finer control.

By migrating or placing specific processes we can make assumptions about them. Specifically, it would normally be necessary to transfer all volatile data from one machine to another for a migration (in UNIX typical data moved would be the stack segment, the data segment and so on). By migrating a known process we can make assumptions about its state. For instance, not all of a process' data segment is going to change during its lifetime. Some parts of the data can be derived from other parts. Other pieces of the data, for instance temporary buffers, are completely unimportant and needn't be migrated at all. If enough is known about the process we can determine which important parts of the data change and send only the differences from some 'default' state.

3.1.2 Migration Timing

Another advantage with migrating a known process is that it is easier to choose a good time to migrate it. State information internal to a process can be used to determine:

- Whether the process is likely to generate a significant amount of work in the near future. Migrating processes creates work for the system. It is only desirable to migrate a process if it is likely to generate additional work of its own in the near future, otherwise the balance of the system will not improve and the work done in the migration process will be wasted.
- Whether a significant amount of work will be required to migrate the process. At various stages in its lifetime a process will have varying amounts of state that would have to be transferred if a migration occurred. The system can estimate this to some extent for any process since the operating system knows how much memory has been allocated to the process. However at any time varying amounts of this allocated memory is 'unimportant', and is not actually required for migration. A system with internal state information can better determine how much should be migrated.

3.1.3 Enhancing Transparency

Tailoring processes for migration can also help alleviate some of the problems related to migration transparency. Many problems associated with running on a remote environment can be solved more easily from within the migrating process. For instance, file name translation can be performed internally, and there is no need for file handle translation as the files concerned can be reopened and the handles reassigned as migration takes place.

3.2 Using Migration Tailoring

Tailoring processes for migration is not a simple task. As a result, we want to tailor as few processes as possible, yet arrange for a significant proportion of the workload to be migrateable. It is not necessary for *all* of the workload to be mobile; we only need enough movement available so that the *excess* load can be moved from one machine to another.

In any event, we are only interested in processes that

- generate a significant amount of work; and
- may be migrated without an excessive amount of effort.

The first point can cause some problems. A tailored version of an application will only generate work if it is actually in use. Different people have different preferences for many applications, while others do not use certain types of applications at all.

For the system discussed in this report, two primary candidates for tailored migration were considered:

- the `twm` window manager; and
- the `tcsh` shell.

The key reason that these were considered is that they are generators of work. In themselves they are not computationally intensive. For instance, the bulk of the work performed in handling a windowing system (screen bitmap redraws for instance) is performed by the X-terminal display program rather than the window manager. The shell program spends most of its time waiting for user input, and is not required to perform intensive computations even when commands are entered. These two programs are however used to start many work-intensive jobs. The window manager starts new processes whenever programs are selected from the backdrop menus, and the shell starts new processes as a result of most commands.

These two programs both start their children on the machine on which they are currently running. As a result, if they are moved from machine to machine, any new processes they create will appear on different machines. Effectively this is a form of process placement, but without the cost of setting up a remote execution environment, something which is computationally expensive and noticeably slow.

For this project the `tcsh` shell program was chosen for modification. While it is not used universally, enough work is generated by it to allow load balancing.

3.3 Load Balancing with the Migrating Shell

The load balancing design presented in this report separates the policy and load moving components of the system into distinct parts, as is common in load balancing systems.

The policy module of this system overall has little to distinguish it greatly from any other load balancing system that has previously been implemented. All of the general policy issues outlined in chapter 2 apply to the policy module of the `tcsh` load balancing system. In the case of the policy used here, standard policy decisions have been made: the policy is a sub-optimal one, based around the centralised load information provided by CLB, and process exchanges always begin as a result of action by the controller on the heavily loaded shell.

The system should be equally applicable to both workstation and processor pool environments, as the a migrating `tcsh` can be used to provide a form of process placement on either. Naturally under a workstation environment difficulties may occur if a user reclaims their workstation while guest shell and children of that guest shell are present. These difficulties are the same as those faced by any idle-processor allocation scheme.

The system does differ however in the interactions between the policy module and the load movement module, and also in the way the load movement module operates. As mentioned above, the system is designed to provide a new form of granularity which will result in a nearly cost-free implementation of process placement.

3.3.1 Transparency Issues in the Migrating Shell

The tailoring of the migrating shell or in fact any process allows the circumvention of many of the problems associated with traditional process placement and migration. Arranging for filename transparency is not difficult: in the case of the implemented example the migrating `tcsh` is designed to work on top of NFS, the Network File System that provides an approximately identical file system interface across all processing nodes on the network. Even if NFS is not present, it is still easier to allow transparent migration. This can be acheived by having the tailored program internally account for differences in its file space.

The user interaction part of the environment can still be difficult to maintain, particularly if the process being tailored has the ability to create children that are not 'migration-aware'. In the case of migration tailoring, and for the shell in particular, interactive input is a necessity and must be supported, unlike in some other systems.

Handling heterogeneous processing nodes in a system is also possible using the migration tailoring method. In the case of `tcsh` some transparency would be lost, since not all programs would run as expected. Transparency could be improved for this by making the shell use different programs to perform the same task on different hosts. In this particular test system we will only be concerning ourselves with (near-)heterogeneous systems.

Chapter 4

Implementation

As far as the t-shell load balancing system is concerned, the movement of work is achieved through the migration of a t-shell (the *local shell*) from the machine it is currently running on (the *local machine*) to some other machine (the *remote machine*). This movement of work is directed by the controllers, which form the policy part of the system.

Therefore the `tcsh` migration system is made up of two parts (as shown in figure 4.1):

- The migrating shell, which forms the load-moving part of the system. The migrating shell is a version of the `tcsh` shell specially modified to allow it to migrate from machine to machine.
- The controller, which performs two functions: maintaining the system-wide loading information and directing the behaviour of the shells on each machine. One controller runs on each of the machines in the network that is participating in the load balancing scheme. The controllers are the only part of the system that routinely communicate across the network.

The components of the system are described in more detail below.

4.1 The Migrating Shell

To migrate a running `tcsh`, the load balancing system must:

- Start up a new t-shell on a remote node;
- Package up the state information of the shell at the local processing node;
- Transfer the state information to the newly started shell on the remote node;
- Unpackage the transferred state information; and
- Make the remote shell the active one.

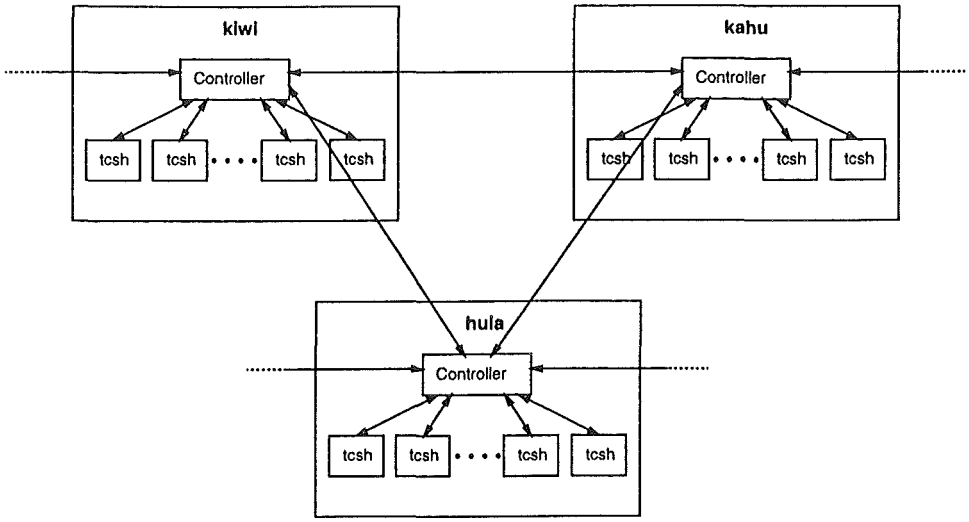

Figure 4.1: The overall organisation of the system. One controller runs on each machine and exchanges load information with peer controllers on other machines. Each controller is responsible for the shells running on its machine.

In the prototype migrating `tcsh` described in this report, the remote shell was started up using the UNIX command `rsh(1C)`. This form of remote execution can cause some difficulties.

The alterations required demonstrate one of the conflicts inherent in this approach: in modifying a program for global accessibility some of the benefits of modularisation can be lost. Some modularisation can be retained by having a separate data packaging routine for each logical section of code. Unfortunately this increases the complexity of the migration system. As a result the migration system can itself become diffuse and potentially difficult to distinguish from within the overall program structure.

4.2 The Current State of the System

The current migrating version of `tcsh` allows manual migration to other machines in response to the command:

```
smigrate xxxx
```

where 'xxxx' is the name of the machine to be migrated to.

At the time of writing, the controller network system has not yet been integrated with the shells. As a result, the load balancing system as a whole is presently inoperative.

However the primary component of the system, that which is being investigated, is the tailored program. This is operational and enough insight has been gained during its construction to establish the issues involved in the creation of a full load balancing system based on this approach.

In the implementation of a tailored program load balancing system such as this one, a number of these issues that would not arise in more traditional migration systems appear. Some of these are detailed below.

4.3 Transferring State

The form of state information transferred over the network is determined by the process that is being migrated. In the case of the migrating t-shell, this essentially consists of all of the variables and lists that can change from when the shell is first started to when the shell is migrated.

Working out which items must be transferred can be a time-consuming and tedious process, and requires a good knowledge of the overall structure of the program being modified for migration. All 'important' variables must be stored and all data structures contained within them must be 'flattened' so that the data can be sent over the network. In addition, under UNIX it is also necessary to transfer the environment variables. In this implementation these are maintained internally by the shell, as the shell itself regularly needs to alter them — as a result it is a fairly trivial task to transfer them.

A summary of the data (state information) transferred follows:

- Environment variables (e.g. `USER`, `TERM` etc.)
- Current Working Directory
- File creation mask (`umask`)
- '`nice`' value
- Shell variables
- Shell aliases
- Shell histories (both of them — see chapter 5)
- Directory stack
- Internal shell state variables (e.g. terminal settings, command line options etc.)

The design choices made in the construction of the program being modified for migration have a large effect on this part of the modification process. Some parts of the structure of the original program must be modified to allow migration to take place. An example is that C 'static' variables are not permitted in a program to be migrated, since it is necessary to extract these values in preparation for migration. These instead become global variables.

4.3.1 Initiating Migration

An important issue where the system being discussed here differs from others is that of communicating the intention to move work from the policy system of the load balancer to the migration mechanism.

In the system presented here the migration system is broken up into many small, separate, discrete parts. To move work, there must be a communication path between the policy-making part of the system and the part that moves the work about. In most other systems this communication is trivial: the load-shifting system is normally all in one place (so easy to assign an address to), or perhaps even a part of the the program that organises the policy, in which communication is even easier.

Instead what we are presented with is a problem similar to that faced by migrating shells trying to contact each other. It is difficult for any outside party, in this case the controller, to contact any one shell, since it has no established address. This problem has been tackled by having the shells contact the controller at regular intervals, easily done since the controller does have a well-known and reachable address.

Shells 'poll' the controller periodically when they consider themselves to be in a state in which migration can be performed effectively. In this system, the shells poll under the following conditions:

- The shell has recently completed execution of a command,
- No processes are running under the shells job control system, and
- This particular shell has not been migrated in the last minute.

Polling in this manner has a number of incidental benefits, namely that it is no longer necessary for the controller (the policy-making component of the system) to concern itself with which parts of the workload should be moved. Shells eligible for movement present themselves to the controller as they become ready for migration, so the controller need make no distinction between them. As long as there is a reasonable number of shells without background processes running on the machine, the controller will receive a steady stream of polls from migration candidates. Note that once again we are adjusting the granularity of the migration system, trading additional control for reduced overheads.

4.3.2 Making Contact

An interesting and unexpected difficulty that did occur in the development of the migrating shell was that of setting up a communication channel between two shells. When a shell is migrating, it must first start another shell on a remote machine, then it must establish a communication channel with the new remote shell.

The current standard protocol available for communicating over networks is the Internet protocol, which is suitable for communicating across differing networks: from small local area networks to the entire network.

The Internet protocols are primarily based around the client-server paradigm: most connections made are made are between servers and clients, where each server may be connected to a number of clients. The important point here is that the server must have a so-called 'well-known' address (consisting of a machine address and a port number) at which it can be contacted. This address is normally assigned explicitly by a human. Examples of servers for which this holds include those utilised by `ftp`, `telnet` and `gopher`.

The application being discussed here however does not explicitly involve a server, so there is no 'well-known' address available to be used as a rendezvous for the two shells to arrange communication through. Dynamically allocating a port number (ie creating a port number 'on the fly' at migration time) is also not a simple task either, as there appears to be no acceptable and portable way of generating such an address.

The best solution at this stage appears to be to use a server for mediation. Instead of attempting to initiate communication between the two undistinguished shells directly, the local shell (the one initiating the migration) requests an identifier from a server running on its local machine. The server is unique to that machine, and thus has a well-known address. The local shell passes the identifier and server address to the remote shell. The remote shell then sets up a communication channel with the local server, using the identifier to announce its association with the local shell. The local shell can now send its state information to the local server, which passes it on to the remote shell.

4.3.3 Terminal Propagation

When a shell migrates, it is important to make this act as invisible to the user as possible. The new remote shell should act like the old local shell in every possible way, including prompting for commands in the same terminal as the old local one.

Propagation of the terminal in this way is an important part of several remote execution programs under UNIX, including `rsh`, `rlogin` and on. `telnet` also performs a similar function.

Ideally, it would be possible to use one of these to interact with the remote shell. However, `rsh` (the ideal choice) does not in fact propagate terminal input and output in an adequate manner. on would be suitable, but it is not a 'secure' program and as such is not installed on many systems. The only standard system available is `rlogin`.

`rlogin` allows very little control over the remote session it starts. As a result, it is rather difficult for the remote shell to determine that it is in fact the destination end of a migration as opposed to a shell being started by a user in the normal fashion. It is also difficult for the remote shell to establish the rendezvous point for connection to the remote shell — this was otherwise passed as a part of the command line arguments to the shell.

The prototype version of the migrating shell developed does not solve this problem. Currently when it migrates the migrated `tcsh` session appears in a new `xterm` window. This is not at all transparent, but allows experimentation with the migrating shell and is adequate for performance testing.

It has been envisaged that the rendezvous between local and remote shell could be arrived at through controller servers, in a manner similar to that described previously. A shell that is about to migrate informs the controller on the remote machine that a new remote shell is soon to be started there. All shells as they start up make contact with their local controller-server to find out whether they are migrated shells, in which case the state transfer process can be initiated. Such a shell could be identified within the controller by the username of the user that is starting it: if a certain user's shell is migrating and a new shell starts up on the nominated destination machine, then it is likely (though unfortunately not certain) that this new shell is the one started by the `rlogin` from the migration initiator (the local shell).

This method of ensuring terminal propagation clearly has some drawbacks. If the user explicitly starts their own shell at approximately the same time as a migration destination shell is expected to appear, then their new shell (started on another terminal) may be mistaken for the migration destination, with confusing results. The user may find that a request for a new shell (say via an explicit `rlogin`) results in a new terminal window with a migrated shell (their original work session) in it, while the old terminal has the fresh session in it — the sessions have been swapped around.

Some other methods of initiating communication between two peer shells have been considered. These centre primarily around making what limited modifications are possible to the destination shells environment through the `rlogin` interface. So far none of these show promise.

The ideal solution to this problem is to write a dedicated terminal propagation system of our own — an equivalent to `rlogin` that does allow modification to the environment of the remote destination process that is started. This is a non-trivial undertaking, involving working with details concerning terminal handling, and possibly even more importantly some security issues: the system is useless if it is not secure (as demonstrated by on). A propagation system such as this might be created by modifying the existing `rlogin` and `rlogind` programs.

With such an ideal terminal propagation system, the simplest way to allow communication link-up would be to provide the remote shell with an address and port number for rendezvous with the local shell's controller. A unique identifier (generated by the local shell's controller) can also be provided and can be used by the controller to match the remote shell with its corresponding local shell (see figure 4.2).

Problems such as this will occur in terminal dependent programs. Load generating programs are more likely to require interactive input and output, so these problems are more likely to be of issue than in systems that simply move CPU intensive batch-type non-interactive processes about.

Similar problems may also occur when tailoring programs that run in a windowing environment. In some windowing systems, windows can have a concept of 'ownership' which can be attributed to processes and perhaps even be unique to one machine. If another process (in fact the migrated version of the original) appears it may be difficult to allow it to take control of the window from the original process (which is now defunct).

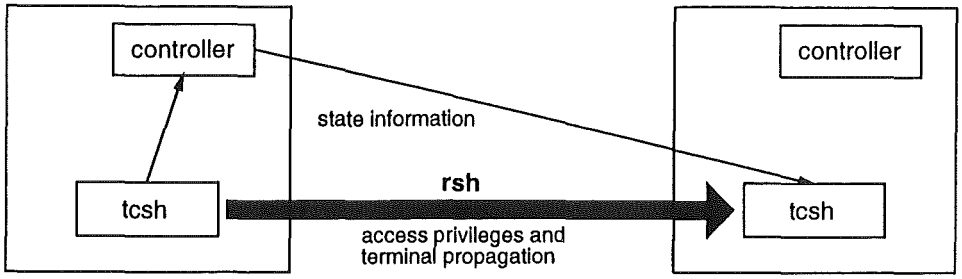

Figure 4.2: Migrating a shell. The local shell registers its intention to migrate with its local controller, and is given an identifier by the controller. The remote shell is started using `rsh`, and is passed the identifier. The remote shell contacts the local controller with the identifier. Finally the local transfers the state information to the controller, which in turn passes this on to the remote shell.

4.3.4 Preventing Dependencies

If a shell migrates then a permanent link must be set up between the new version of the shell on the remote machine and the old version of the shell on the local machine, for the purposes of terminal propagation. If the shell on the remote machine were to migrate again, then under ordinary circumstances it would be necessary to set up another permanent communications link for terminal propagation. Terminal output would flow from the most recent invocation of the shell, to the previous migrated shell back to the shell on the original host before finally being output.

This has a number of obvious disadvantages:

- A normal system is likely to have a number of shells running on it. If each of these shells were to migrate a few times, then it is likely that the system would spend an increasingly large proportion of its time servicing terminal interaction between the various shells and hosts.
- Where a chain of migrated shells exists between the most recent invocation and the original shell, if any one of the in-between shells were to fail (perhaps because the machine on which it was running crashed) then the shells would fail. As a result, the reliability of the system as a whole would be reduced considerably.
- Series of migrating shells created in this way would use up a lot of memory — each shell running on our local system occupies approximately 600K of memory, so if too many invocations appear the system as a whole is likely to run short of memory.

This problem can be eliminated by handling 're-migrations' of this type in the following way (as shown in figure 4.3):

- The current remote shell (ie the one that is migrating) is told to migrate to a new machine. It sends its state information and the new destination

machine to the controller on the machine that is running the original shell (the one that is directly connected to the terminal system). The migrating shell can now die quietly, which immediately informs the originating shell that something is happening.

- The controller on the originating machine tells the originating shell (which now exists purely for the purpose of propagating the terminal) to create a new shell on the next destination machine.
- Migration now proceeds as normal: the originating shell creates a new remote shell, giving it terminal access. The new shell contacts the originating machine's controller and receives the state left there by the most recently migrating shell. It can now process that state information and continue execution.

Using this method, residual dependencies are can be avoided. All traces of the activity of this shell are eliminated from the first migration destination machine at the end of the procedure.

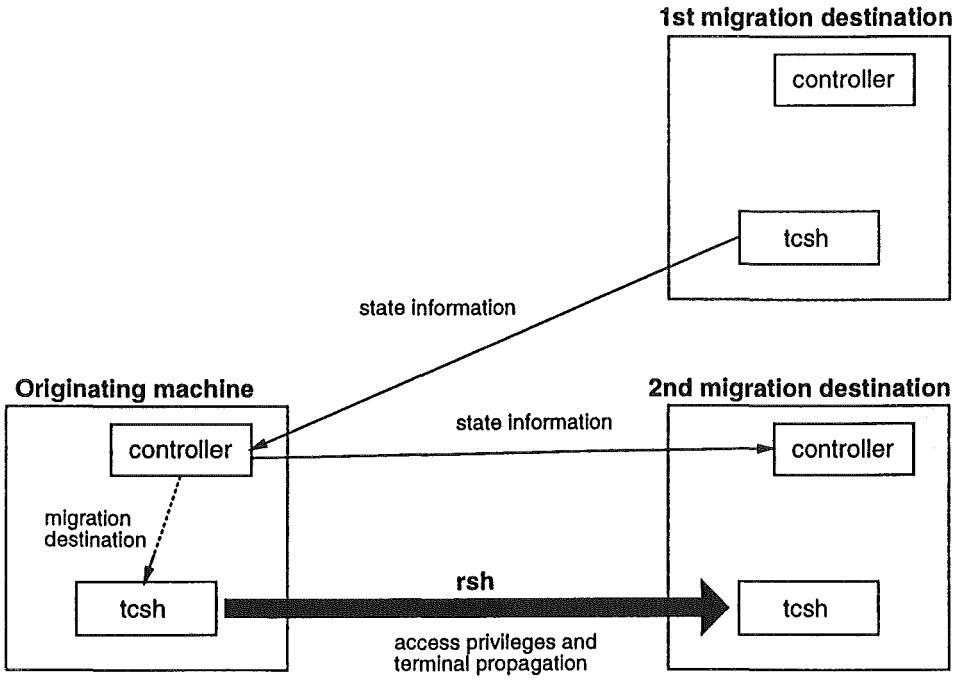

Figure 4.3: Re-migrating a shell.

4.3.5 Interaction With Other Processes

The approach under investigation here involves work generating programs. In reality this will often mean that any program being worked with and modified

will be a program that creates other processes. In some cases it is required that the work-creating program interact with the child processes that it has created.

In the case of the t-shell the child processes involved are those created by the t-shell as it executes user commands. In particular, these are most likely to be processes which are running in the background or have been suspended.

The shell interacts with these programs in its capacity of job controller. The shell has direct control of child processes: it is periodically called upon to kill or suspend its child processes. In addition, whenever any of the shell's children dies, the shell must be informed so that it can update its process tables and inform the user that the process has terminated.

Under UNIX most of this operates through the UNIX signal system. Signals do not propagate across machines. If the shell migrates to another machine it will lose communication with its child processes.

It is impractical to migrate the children along with the shell. If this was attempted, then we would be encumbered with all of the problems associated with standard migration systems. An example of this is that these child processes may themselves have children, or worse still they may be interacting with other apparently independent processes on the local machine.

Instead we are presented with two options: either we can only consider migrating processes that have no associated children (in our case only move shells that have no background processes running), or we can interpose to allow signal propagation communication between the shell (on a remote machine) and its children (on the local machine) across the network.

This could be handled using a 'stub' process that stays in the local shells stead when the shell migrates away. The stub could receive signals when child processes die and send these over the network to the shell in its new location. Correspondingly, shell job control could be handled by the now remote shell sending messages to its local stub representative, which then sends the controlling signals to the child processes concerned.

In the case of `tcsh` it was decided not to introduce the added complexity of signal propagation. Only shells that are controlling no background processes are considered for migration. The rationale behind this is that presented earlier: it is unnecessary to allow all of the work to be migrated, only that proportion required to balance the load. As long as most of the shells on a system are not running background processes (this is normal on our installation, at least), enough active work-generating shells remain to permit effective distribution of work.

4.4 The Policy System

While the policy system has not been the focus of this project, one is nevertheless required. The policy implemented for the t-shell migration system is based around that provided by the CLB system. A primary reason for this is that the t-shell system is designed to work in well with the CLB system. An undesirable possibility that can occur is that of competing systems: if the two systems disagree as to what should be done to correctly balance the load of the

system, then they may tend to work against each other, creating more work for the processors on the network in their efforts to move load about.

The policy system is implemented by the controllers, one of which runs on each of the processing nodes. The load metric developed for CLB is used as a basis for decisions on when and where to migrate.

The decision to migrate processes is made on the basis of the most and least heavily loaded processing node on the network, as judged by the metric. If these differ by more than a predetermined amount (the *offload constant*) the controller on the most heavily loaded machine sends an 'offload' request to the controller of the least heavily loaded machine. This is a check to ensure that the lightly loaded machine has the resources required to support another shell.

If the lightly loaded processing node is prepared to take the additional work, it sends an acceptance message back to the controller on the heavily loaded node. Otherwise a rejection message is returned, with a reason for rejection. If its selected destination is not prepared to accept additional load, the offloading controller tries the next most lightly loaded controller. It will keep on retrying with successively more heavily loaded processors until one accepts or there is no processor available whose load is at least the offload constant.

Once the offload request has been accepted, the two controllers form a "buddy" pair. Until the load exchange between these two is complete, they will not consider or permit load exchanges with any other processing node.

This is done to help prevent instabilities from occurring: should more than one controller attempt to offload work at once, then this will not allow, for instance, multiple controllers offloading work onto one processing node (so making it the next offloader candidate).

An issue that is common to both this system and many others is the decision of how much should be migrated. The shells we are migrating can have highly variable workload-generating characteristics. At the time of migration a shell is idle; it is difficult to determine how much work each shell is going to generate in the near future. Therefore it is also difficult to work out how many shells should be migrated to equalise the loads between the two processors. For our purposes, stability is important — if we "overshoot" by transferring too much work to the destination then *host overloading* (Ferrari & Zhou, 1988) will result, and we may effectively have to move some of that work back off again in the near future (so effectively undoing some of the work in the first migration effort). Instead it is desirable to be conservative in migration — each migration creates work and if we need to we can easily arrange (at little cost) for the offloader to offload more work at a later date.

Chapter 5

Limitations of the Approach and Future Possibilities

5.1 Global Program Usage

One major problem with the tailored migration approach is that for the modified program to be effective in moving work, it must have an active role in the creation and management of work on the system. This fact may sound trivially obvious, but it takes on a whole new meaning when the knowledge that different people use different programs to do the same things is taken into account.

In this report we looked at an example of the tailoring for migration of the `tcsh` command interpreter. Other shells also exist (eg `sh`, `csh` and `bash`), and some people use these in preference to tcsh. If these people continue to use their different shells then the work they generate will not be available for movement by the load balancing system.

Some undergraduate users in our local environment here at the University of Canterbury do not in fact use a standard command-line shell at all. Instead all of the work they generate is created through the manipulation of a graphical user interface, which never interacts with the migrating `tcsh`.

These are examples of the increasing diversity of programs present in user environments. This is likely to continue in the future, and as a result it is more difficult to find a tailorable program responsible for a large proportion of the workload on a system now than it would have been ten years ago.

5.2 Future Work — Utilising Public Histories

A suitably modified version of `tcsh` has successfully been created for the purpose of load balancing in a moderate amount of time, relative to implementation times of other load balancing systems. This has shown that the implementation of load balancing systems based on this approach is indeed practical in view of the potential benefits. The most important thing that can now be done is to quantify the benefits of the load balancing system. A framework for this testing was created as a part of the familiarisation process when shell modifications were commenced.

A problem with performance evaluation of past load balancing systems is that it is difficult to test them with realistic loading patterns. Either researchers have opted to used fixed artificial benchmark workloads to test the system (Ferrari & Zhou, 1988), or researchers have used repeated experimentation in real-life workloads to test system performance.

The main problem identified with the first testing method is that it is difficult to simulate the complexities of real usage in a network operating system using an artificial load. The second testing method is tested under real-life conditions, but these conditions are variable: when an alteration is made to the system it is difficult to show that changes in the test results are in fact a direct result of the system alterations made.

The testing system is based around the so-called 'Public Histories' which have been implemented in the modified version of `tcsh`. The Public History system allows the work-generating characteristics of all `tcsh` users (those who permit it) to be stored in a central repository. This repository contains the resource usage of all users' commands, and can be used to make more realistic test loads for use in performance simulations. Metrics made in such a way can still not be entirely realistic, due to the fact that not all of the load on a system is generated by shells. Nevertheless, it should be an improvement over completely artificial workloads.

Chapter 6

Conclusions

In many ways the tailoring for migration method of load balancing can be considered to be a compromise between the process placement method and the user login placement method of load balancing. It has the finer granularity and finer control of process migration or placement, without the full costs of either. Unfortunately it is considerably more complex to implement a tailored migration system than it is to implement a user placement scheme and the effectiveness of the system will depend to a larger degree than normal on the regular usage patterns of the users.

The greatest drawback of this method is the effort required to successfully tailor a program for migration. For the tailoring to have any great effect, it is likely that the program to be tailored is going to be a popular one. Popular programs in most installations capable of running load balancing schemes are likely to be large and complex, each with a large number of contributors. This adds to the complexity of the alterations required. The modification process could be simplified somewhat if a 'migration toolkit' were made available for use in migration tailoring applications.

It should be noted that systems like the one introduced in this report that involve modification to existing network operating systems can have a limited lifetime at most, as they will eventually be replaced by full distributed operating systems. However, looking at the history of operating systems it is likely that existing network operating systems will continue to be in use for some time, and while this is so there will be a need for systems like the one described here.

References

Ashton, Paul, & Smith, Peter. 1993. The clb load balancing system. *In: Uniforum New Zealand 10th Annual Conference.*

Barak, Amnon, & Shiloh, Amnon. 1985. A distributed load-balancing policy for a multicomputer. *Software: Practise and experience*, **15**(9), 901 – 913.

Bricker, Allan, Litzkow, Michael, & Livny, Miron. 1992 (Jan.). *Condor technical summary.* Tech. rept. 1069. Computer Sciences Dept, University of Wisconsin-Madison.

Eager, D. L., Lazowska, E. D., & Zahorjan, J. 1986. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, **6**(1), 53 – 68.

Eager, D. L., Lazowska, E. D., & Zahorjan, J. 1988 (May). The limited performance benefits of migrating active processes for load sharing. *Pages 63 – 72 of: Proceedings of the 1988 ACM SIGMETRICS conference on measurement and modelling of computer systems.*

Ferrari, Domenico, & Zhou, Songnian. 1988. An emperical investigation of load indices for load balancing applications. *In: Performance '87.* North-Holland.

Goscinski, A. 1991. *Distributed operating systems: The logical design.* Addison-Wesley.

Hac, Anna. 1989. Load balancing in distributed systems: A summary. *Performance evaluation review*, **16**(2), 17 – 19.

Nichols, David A. 1987. Using idle workstations in a shared computing environment. *Pages 5 – 12 of: Proceedings of the eleventh ACM symposium on operating system principles,.*

Smith, Peter. 1992. *Investigation of load-balancing for a network of Suns.* Honours project report. Department of Computer Science, University of Canterbury.

Stevens, W. Richard. 1990. *UNIX network programming.* Prentice-Hall.

Tanenbaum, Andrew. 1992. *Modern operating systems.* Prentice-Hall.

Theimer, Marvin M., Lantz, Keith A., & Cheriton, David R. 1985 (Dec.). Preemptable remote execution facilities for the v-system. *Pages 2 – 12 of: Proceedings of the tenth acm symposium on operating systems principles.* Operating Systems Review, vol. 19, no. 5.